

# 1. 本周工作

阅读《C++ concurrency in action》Chapter 4. Synchronizing concurrent operations（内容提炼见附录）。

# 2. 下周工作

阅读Chapter 5. The C++ Memory and operations on atomic types

## 附录

### Chapter 4. Synchronizing concurrent operations (同步并发操作)

本章主要包括：

- 用条件变量等待一个事件
- 用future等待一个事件
- 有限时间的等待
- 用同步操作化简代码

一个线程P1在移进（完成自身任务）前，必须等待另一个线程P2完成任务，是一个常见的情景。

更广泛地说，一个线程等待某特定事件的发生或某个条件变为真，是非常普遍的现象。

定时查询可能可以实现这个功能，但远远不够理想，C++标准库提供了condition variables和futures来完成这个任务。

#### 4.1 Waiting for an event or other condition

解决P1必须等待P2的同步问题，有三种方法。

方法一，P1轮询一个由mutex保护的共享数据(flag), P2负责在完成自身任务后设置这个flag. 这种方法的两个缺点是：

- P1轮询浪费了宝贵的CPU时间
- P1轮询时持有锁，导致别的线程无法获得该锁，降低程序并发性，影响了别的线程的效率

方法二，在flag == false时，让P1沉睡一段时间

```

bool flag;

mutex m;

void wait_for_flag()
{
    unique_lock<mutex> lk(m);

    while (! flag)

    {
        lk.unlock();

        this_thread::sleep_for(chrono::milliseconds(100));

        lk.lock();
    }
}

```

但难以恰当设置P1的沉睡的时间：

- 设置得太短，频繁唤起检查flag无疑是对CPU时间的浪费
- 设置得太长
  - P1可能进行了无谓的等待，导致它响应速度的下降
  - 考虑别的线程P3在等待P1的情况，P1本可以继续进行，但却仍处于sleep状态，进而影响等待着它的进程P3的执行效率

方法三，用C++标准库提供的条件变量（condition variables）实现一个可被别的线程触发的事件等待。

#### 4.1.1 Waiting for a condition with condition variables

```

mutex mut;

queue<data_chunk> data_queue;

condition_variable data_cond;

// P1

void data_preparation_thread()
{
    while (more_data_to_prepare())

```

```

    {
        const data_chunk data = prepare_data();

        lock_guard<mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}

// P2, 3, 4...
void data_processing_thread()
{
    while (true)
    {
        unique_lock<mutex> lk(mut);
        data_cond.wait(lk, []{ return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();

        lk.unlock();

        process(data);

        if (is_last_chunk(data))
        {
            break;
        }
    }
}
}

```

P2, 3, 4...等待P1生成新的数据。

`data_cond.wait(lk, [] { return xxx;})`的原理是：

- 在第一次调用时，检查条件（判断第二个参数（callable object）的返回结果）
  - 返回true, 则继续执行

- 返回false, 则释放锁, 将线程编程阻塞或等待状态 (puts the thread in a blocked or waiting state)
- 当线程被P1-notify\_one()选中唤醒时, 会重新加锁, 并检查条件
  - 返回true, 则继续执行
  - 返回false, 则释放锁, 将线程编程阻塞或等待状态

正是由于在一个线程中多次加锁解锁, 所以采用灵活性更强的unique\_lock而非lock\_guard:

- wait()中可能需要加解锁
- process(data)可能是个耗时的操作, 在这个操作前解锁同样可以增强并发性

#### 4.1.2 Building a thread-safe queue with condition variables

忽略构造、赋值和交换函数, queue拥有下列三类操作:

- 查询队列状态(size(), empty())
- 查询队列元素(front(), back())
- 改变队列状态(push(), pop(), emplace())

这和3.2.3中介绍的栈一样, 在不引入外部变量的情况下, 先检查后操作的push(), pop(), emplace()都将引发**race conditions inherent in the interface**.

本节实现了采用了mutex及condition\_variable作为成员变量的线程安全类threadsafe\_queue.

如果只需要等待线程一次: 一旦wait()等待的条件为真, 该线程将不再需要等待该条件变量, 那么条件变量可能不是最好的实现机制。当线程等待的是一份在条件为真时传来的数据时, 这个观点尤其正确。这是future可能会是更好的选择。

### 4.2 Waiting for one-off events with futures

#### 4.2.1 Returing values from background tasks

头文件中的async函数将启动一个异步任务用来计算不是立即需要的结果。与返回一个需要等待 (可能指的是join) 的thread对象不同, async返回的future<>对象最终将拥有计算结果。

当需要这个结果时, 可以调用future的成员函数get(), 当前线程将阻塞直至future对象准备好(until the future is ready)并返回结果(and then returns the value), **future<>的模板参数为其返回结果**。

本小节介绍了future任务的传参, 在此不予以展开。

async()不是唯一一个将future与一个任务关联的方式, packaged\_task<>和promise<>都可以实现这一点。由于packaged\_task<>比promise<>更高层, 我们先介绍packaged\_task<>。

#### 4.2.2 Associating a task with a future

packaged\_task<>将future与一个函数或其他可调用对象绑定 (ties), 模板参数是函数签名。

当packaged\_task<>对象被调用时, 它会调用相关的函数或可调用对象, (最终)使得future对象就绪。

相关的future对象可以通过obj.get\_future()获得。

本小节给出了用packaged\_task<>及future<>实现的gui框架。

#### 4.2.3 Making (std::) promises

promise提供了设置值的方法，供它对应的future(obj.get\_future())读取。模板参数是值的类型。

promise/future对为如下机制提供了基础：

- 等待线程P2, P3, ...可以在某个future上阻塞
- 提供数据的线程P1可以设置promise的值(set\_value(arg))，以使得future就绪，唤起线程P2, P3等

无论是packaged\_task还是promise都可能抛出异常，下一小节介绍异常的捕获与处理。

#### 4.2.4 Serving an exception for the future

考虑由async引发的函数调用产生了异常，这个异常，而非期待的返回结果，将被存储到对应的future中。

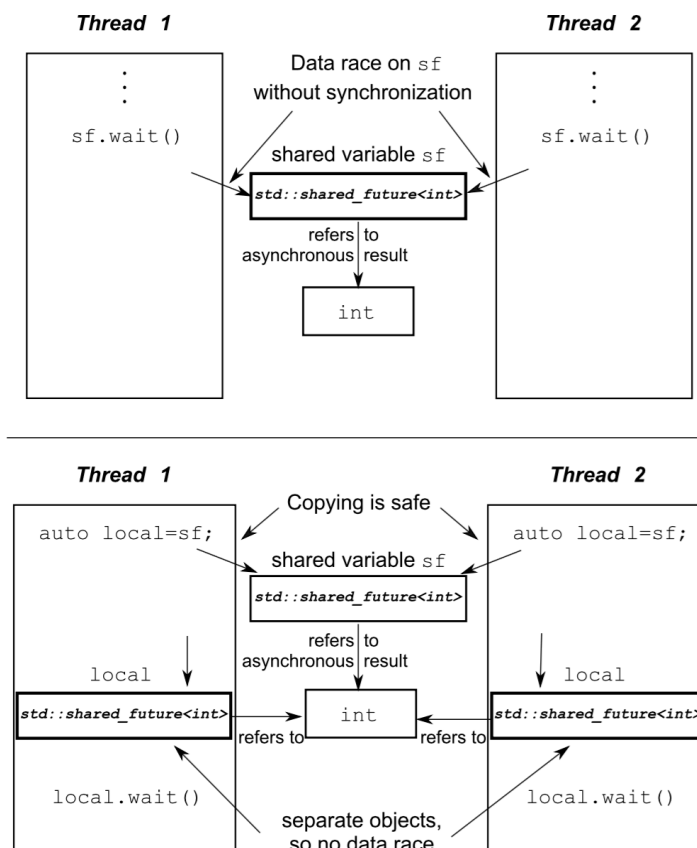
在future对象调用get函数时，这个异常将再次被抛出。但标准没有描述抛出的是这个异常本身，还是它的副本。

#### 4.2.5 Waiting from multiple threads

future同步了两个线程间的数据传输，却没有对它自身的成员函数进行同步——多个线程访问future对象时可能引发数据竞态。future的设计决定了，只有一个线程可以通过get()获得future对象的返回值，其他的get()的调用都是没有意义的。

future是不可复制可移动的，但shared\_future是可复制的。

shared\_future的成员函数仍非线程安全，但可以复制其对象，在新的对象上调用成员函数，访问共享的异步状态，或尝试获得计算结果，如下图所示：



**Figure 4.1 Using multiple `std::shared_future` objects to avoid data races**

```
shared_future<string> sf1(promise1.get_future()); // method1
auto sf2 = promise2.get_future().share();        // method2
```

法1法2都实现了shared\_future的构造:

- 法1隐式转移了future的所有权
- 法2通过share()函数, 构造了新的shared\_future对象, 并转移了future的所有权。法2选用auto定义对象, 减少了编码工作, 并使代码更容易维护、修改。

## 4.3 Waiting with a time limit

之前介绍的阻塞调用都可以阻塞无限长的时间, 将线程挂起直至等待的事件发生。

本章将引入限时的阻塞调用, 这些调用基于下述两种限时方式:

- a duration-based timeout, 对应带有\_for后缀的函数调用
- an absolute timeout, 对应带有\_until后缀的函数调用

### 4.3.1 Clocks

clock作为类, 提供了四种信息:

- now, 当前时间
  - e.g., std::chrono::system\_clock::now();
- time\_point, 表征从clock类获得的时间的类型
  - e.g., some\_clock::now()的返回值应该是some\_clock::time\_point
- period, clock每次tick之间的秒数
  - e.g., 一个每秒ticks25次的时钟, 它的period将是ratio<1, 25>
  - 若ticks间隔必须等待运行时才能确定或在运行时将变化, 那么period的值可能是: 这些间隔的平均值、最小值或库作者认为合理的其他值。
- is\_steady, 静态方法, 用于判定是否是稳定时钟
  - 稳定时钟的定义: ticks间隔不变(无论是否等于period), 时钟不可以被调整(can't be adjusted)
  - std::chrono::system\_clock不是稳定时钟: 它可以被调整

### 4.3.2 Duration

std::chrono::duration<type, seconds>, 持续时间:

- type: 值的类型(int, long, double)等
- seconds: 每个unit代表的秒数
- e.g., std::chrono::duration<short, ratio<60, 1>>可以表征以分钟为单位的持续时间。

标准库在std::chrono命名空间下提供了提前定义好的持续时间类型:

- nanoseconds
- microseconds

- milliseconds
- seconds
- minutes
- hours

duration间的转换：

- 在不需要truncation的情况下，不同的duration可以进行隐式转换
- 否则需要显示转换：

```
std::chrono::seconds s = std::chrono::duration_cast<std::chrono::seconds>(ms);
```

这种转换采用truncation, 而非rounding

所有等待函数(wait functions)都会返回一个状态用以表明超时或等待时间已发生。例如，在等待一个future时，返回：

- std::future\_status::timeout
- std::future\_status::ready
- std::future\_status::deferred, future的task被延迟了

### 4.3.3 Time points

std::chrono::time\_point<clock, duration>:

- clock, 时钟类型
- duration, 持续时间类型

time\_point<>实例的值代表自clock的epoch起的时间长度。

time\_point<>支持：

- 在某个实例上，加上一个duration.
- 对共享相同clock(shares the same clock)的时间point实例进行减法，获得duration

下述是对条件变量进行限时等待的代码示例：

```

std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
    auto const timeout = std::chrono::steady_clock::now() +
std::chrono::milliseconds(500);
    unique_lock<std::mutex> lk(m);
    while (!done)
    {
        if (cv.wait_until(lk, timeout) == std::cv_status::timeout)
        {
            break;
        }
    }
    return done;
}

```

根据[www.cplusplus.com](http://www.cplusplus.com)提供的手册，我认为while(!done)的用处在于：

- wait\_until(lk, timeout)被调用后，函数可能在被notified或超时被唤醒，它被唤醒时都会返回值，所以需要不断循环，直至它返回std::cv\_status::timeout.

#### 4.3.4 Functions that accept timeouts

列表简述了支持限时等待的函数。

### 4.4 Using synchronization of operations to simplify code

我认为C++支持的编程范型包括但不限于：

- 命令式编程
- 面向对象编程
- 泛型编程
- 函数式编程 (FP, Functional Programming)
- 通信顺序处理编程 (CSP, Communication Sequential Processes)

本章介绍了FP和CSP在C++多线程编程中的应用。

#### 4.4.1 Funtional programming with futures

函数式编程描述的是一种函数输出与外部状态无关，只依赖于输入的编程方式。

纯函数不会改变外部状态，它的影响仅限于其返回值。

本小节使用future的函数式编程实现了快排算法。

#### 4.4.2 Synchronizing opearitions with message passing

这是Erlang和MPI(Message Passing Interface)采用的范例(paradigm).

真正的CSP没有共享数据，所有通信都靠消息队列传递，线程只需要考虑如何响应它接受到的信息，等效



于一个状态机。

因为C++的线程共享了地址空间，不可能强制实行这个标准，所以需要应用程序或库作准遵循一个准则：在线程间不共享数据。